Peer-to-Peer Federated Learning
A Decentralized Approach

Geetha Sowmya Chitneedi - CS20BTECH11011

Venkata Datta Sri Harsha Kusampudi - CS20BTECH11028

May 2, 2024

Contents

1 Abstract

Federated learning has emerged as a promising approach for collaborative model training across decentralized edge devices while preserving data privacy. But central servers represent single points of failure, making the system vulnerable to attacks or outages. To address this issue, this report introduces a novel peer-to-peer (P2P) federated learning framework, where network nodes directly communicate with each other, facilitating distributed model training without the need for a central server. We investigate both synchronous and asynchronous systems, presenting algorithms and providing justification for their selection. Additionally, we conduct comprehensive experiments to assess the framework's performance, communication overhead, and time efficiency relative to varying number of client nodes and density of the networks. Our findings shed light on the advantages and limitations of P2P federated learning compared to traditional server-client architectures. The code can be found here

2 Introduction

Federated learning has emerged as a promising paradigm for training machine learning models across distributed devices while preserving data privacy. In this context, we present a novel peer-to-peer federated learning framework designed to overcome the limitations of traditional server-based architectures. Our framework leverages an undirected graph topology, wherein each node acts as both a client and a server, enabling seamless communication and collaboration among peers. Unlike centralized approaches, our framework operates without a single global model, preserving data privacy by only exchanging model parameters during collaboration. We proposed the methodology of our framework, highlighting the federated averaging algorithm used for parameter aggregation in both synchronous and asynchronous systems. Additionally, we analyze the framework's crash tolerance, demonstrating its resilience to node failures. Finally, we present experimental results evaluating the framework's performance in terms of model accuracy, communication cost, and time efficiency across various network densities and client numbers.

3 Related Works

Communication-Efficient Learning of Deep Networks from Decentralized Data[3]: This paper presents an approach to federated learning where deep neural networks are trained on decentralized data sources without requiring raw data to be centralized. By leveraging local computation on individual devices and exchanging model updates rather than raw data, the communication overhead associated with traditional centralized learning approaches is significantly reduced. This paper laid the foundation for research in federated learning, highlighting its potential for applications in distributed environments such as edge computing and IoT networks. This approach is fundamental to the concept of federated learning, where privacy and data locality are paramount concerns.

Federated Learning With Quantized Global Model Updates[1]:
In this paper, building upon the foundational principles of federated learning, they introduced a novel approach to further enhance communication efficiency in decentralized learning systems. In this work, the authors proposed the use of quantization techniques to compress model updates exchanged between devices during the federated learning process. Specifically, they introduce an aggregation method called FedAvg, which aggregates model updates by taking the average of parameters across devices; this method effectively reduces the communication overhead without sacrificing learning quality. The paper provides insights into the trade-offs between communication efficiency and model accuracy in federated learning.
Complexity of Network Synchronization[2]:
This paper by Baruch Awerbuch, published in the Journal of the ACM, is a landmark work in the field of distributed systems, focusing on the intricate problem of network synchronization. The concept of network synchronization is fundamental in ensuring coordinated behaviour among nodes in a distributed system, even when they operate asynchronously and face communication delays or failures. The paper addresses the complexity of achieving network synchronization and introduces the notion of an α synchronizer. The α synchronizer works as follows: Before incrementing the tick, each node needs to ensure that it is safe to do so. The α-synchronizer implements this by asking each process to send a $\langle safe \rangle$ message after it has sent and received all messages for the current clock tick. Each process executes the following three steps for the simulation tick i:

1. Send and receive messages $\langle m, i \rangle$ for the current tick i.
2. Send $\langle ack, i \rangle$ for each incoming message, and receive $\langle ack, i \rangle$ for each outgoing mes- sage for the current tick i.
3. Send $\langle safe, i \rangle$ to each neighbor. When a process receives $\langle safe, i \rangle$ messages from every neighbor for tick i, it increments its tick to $(i + 1)$ and starts the simulation of tick $(i + 1)$.

These papers collectively contribute to the foundation of peer-to-peer federated learning by addressing key challenges related to communication efficiency, model accuracy, and synchronization in distributed systems.

4 Methodology

In our P2P federated learning framework,

- **Undirected Graph Topology**: The nodes are connected in a network where each node can communicate directly with other nodes. This topology allows for communication and collaboration among nodes.
- **Each node acts both as a client and a server**: Each node in the network serves as both a client because it trains its own model and a server because it can communicate with other nodes to exchange model parameters.

- **There is no notion of a global model**: Unlike traditional centralized learning approaches, there's no single global model to which all nodes contribute. Instead, each node maintains its own local model.
- **All the nodes train on their local models with their local data, which is not shared with other clients**: Privacy is preserved because raw data is not exchanged between nodes. Instead, only model parameters or updates are shared.
- **The trained local models' parameters are used in the collaborative learning process**: Model parameters are shared between nodes to collaboratively improve the models.

We use the federated averaging algorithm [**<empty citation>**] for aggregation of model parameters. Each node exchanges the locally updated weights and averages the weights received, and the averaged weights are assigned back to the model. Now, we will look into our algorithm in synchronous and asynchronous systems.

4.1 Synchronous system

In this section, we assume the system is synchronous, and we propose a peer-to-peer federated averaging algorithm that runs in rounds in a lock-step manner.

Algorithm 1 Federated Averaging on Synchronous System

```
1: Each client $i$ executes the below algorithm
2: Parameters: $K \leftarrow$ Number of local update steps, $R \leftarrow$ Number of Rounds , $local\_data_i \leftarrow$ Data at client i
3: Initialization: $model_i$ is randomly initialized
4: for round $= 1, 2, \ldots, R$ do
5:     for $k = 1, 2, \ldots, K$ do
6:         Train $model_i$ on $local\_data_i$
7:     end for
8:     Broadcast $model_i$'s parameters to its neighbors
9:     Receive $model_j$'s parameters from its neighbors
10:    Initialize $averaged\_model$ as zeros
11:    for all $model_j$ parameters received do
12:        $averaged\_model \mathrel{+}= model_j$
13:    end for
14:    $averaged\_model \mathrel{/}= num\_j\_received$
15:    $model_i = averaged\_model$
16: end for
17: return $model_i$
```

Justification: In the algorithm 1, we assumed the system to be synchronous. So, there is no problem with rounds because all processes are in the same round at any instant. Each node acts both as a client and server. So, the correctness can be attributed to the traditional FedAvg [1] algorithm, where similar things happen, if we assume each node consisting of two parts - server and client, clients of all nodes are similar to clients in centralized Fedavg algorithm. The servers of all nodes are functionally similar to the single server. In this way, the algorithm can be justified to function as expected.

4.2 Asynchronous system

In this section, we assume the system is asynchronous. So, we add an α-synchronizer [2] to make the Algorithm 1 run on this asynchronous system.

Algorithm 2 Federated Averaging on Asynchronous System

```
1: Parameters: $K \leftarrow$ Number of local update steps, $R \leftarrow$ Number of Rounds , $local\_data_i \leftarrow$ Data at client i
2: Initialization: $model_i$ is randomly initialized
3: for round $= 1, 2, \dots$ do
4:     for $k = 1, 2, \dots, K$ do
5:         Train $model_i$ on $local\_data_i$
6:     end for
7:     Broadcast $msg_i = \langle$model$_i$'s parameters, round, $i\rangle$ to its neighbors
8:     Receive $msg_j = \langle$model$_j$'s parameters, round, $j\rangle$ from all its neighbors
9:     for each $msg_j$ received do
10:        Send an acknowledgment message $\langle$ ack, round, i $\rangle$ to j
11:    end for
12:        Receive acknowledgments for all $msg_i$ sent
13:    Initialize $averaged\_model$ as zeros
14:    for all $model_j$ parameters received do
15:        $averaged\_model$ += $model_j$
16:    end for
17:    $averaged\_model$ /= $num\_neighbors_i$
18:    $model_i = averaged\_model$
19:    Broadcast safe message $\langle$ safe, round $\rangle$ to its neighbors
20:    Receive safe message $\langle$ safe, round $\rangle$ from all its neighbors
21:    Move to the next round
22: end for
23: return $model_i$
```

Justification: If the system is synchronous, the algorithm 1 can be used. So, we try to make the system synchronous by adding a synchronizer on the top of the asynchronous system. The correctness of this async algorithm 2 can be attributed to the correctness of α synchronizer. So, by adding the α-synchronizer, we can assume that the rounds happen synchronously. Now, in this synchronized system, as we used the same logic as in 1, the correctness of the algorithm can be justified.

4.3 Crash tolerance Analysis

In a synchronous system, our approach withstands against crash failures. In the traditional server-based approach, the server is prone to be a single point of failure, i.e., if the server crashes, the whole learning stops until the server is again restored. The main advantage of our P2P approach is that there is no such single point of failure. For example, if a node crashes during the training, the other nodes can detect this using mechanisms like timeouts, and they can thus communicate only with the neighbours who are alive. This adaptability ensures that the learning process continues seamlessly despite individual node failures.

5 Experiments and Results

5.1 Data Description

We created synthetic data with 1000 data samples (x, y) where $x \sim 10 * N(0, 1)$, according to the relation $y = 3x + 4 + \epsilon$ where $\epsilon \sim N(0, 1)$. We took the train set, val set, test set in the ratio $70 : 15 : 15$.

5.2 Experimental Setup

We considered a fully connected feed-forward neural network with one hidden layer of dimension 10. The models used for testing are those with the best validation set loss, which is calculated after each epoch. The asynchronous algorithm 2 is implemented for training the model.

Design Implementation of P2P architecture:
We assumed the communication to happen in 4 forms of messages

1. model_msg $\rightarrow \langle$ model_msg, sender_id, round, model_parameters $\rangle$
2. ack_msg $\rightarrow \langle$ ack, sender_id $\rangle$
3. safe_msg $\rightarrow \langle$ safe, sender_id $\rangle$
4. marker_msg $\rightarrow \langle$ marker, sender_id $\rangle$

Each process/node is assumed to be running two threads in parallel.

1. trainer $\rightarrow$ responsible for doing local model updates and aggregation of the models.
2. receiver $\rightarrow$ responsible for receiving messages.

Trainer thread: The trainer thread of node i, initializes the $model_i$. For r rounds, the following process is repeated. The $model_i$ is trained for k steps using the $data_i$, which is local to node i. It is busy waiting for all the neighbours to send their model parameters. Once it receives the model parameters from all neighbours, it updates the $model_i$ using the FedAvg rule. The validation loss is calculated, and the best models are saved after each epoch. It is busy waiting till all the neighbours send acknowledgements for the model messages sent. After receiving all acknowledgements, it sends a safe message to all its neighbours. Once it gets the safe message for round r, the trainer progresses to the next round. After completing all rounds, it sends a marker message to all its neighbours and terminates.

Receiver thread: The receiver thread of node i, receives the messages by listening on a socket which is bind at some fixed address. If the received message is a model message, the receiver thread stores the model parameters received and sends an acknowledgement to the sender. Otherwise, for the other messages, it updates the corresponding counts for each type of message in a common variable shared with the trainer so that it can perform the actions as needed. After receiving markers from all neighbours, this thread terminates.

Design Implementation of Server-Client architecture:

The server node has two threads

1. server_trainer $\rightarrow$ responsible for aggregating the model parameters received from clients.
2. server_consumer $\rightarrow$ responsible for receiving the updated parameters from the clients.

Server trainer thread: This thread does the following: For r rounds, it sends the initial mode to all the clients. Once the client sends the updated model parameters, it aggregates them according to the FedAvg rule and moves to the next round.

Server Consumer thread: Receives and stores the model parameters sent by the clients.

The client node has **client_trainer** thread, which receives the model sent by the server and performs k local update steps on its local data, sending the updated model back to the server.

For all the experiments, the number of rounds = 10, number of epochs = 10.

5.3 Performance Comparison

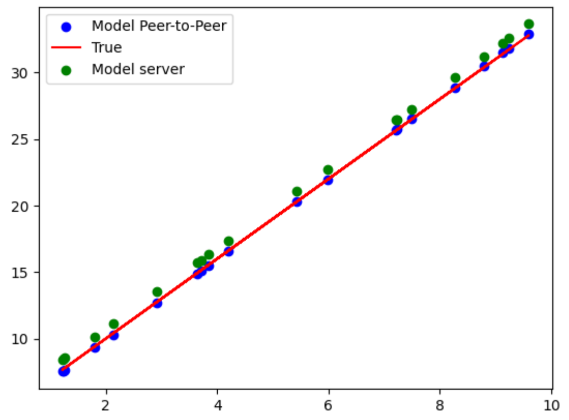

Figure 1: Predicted target vs Ground Truth for Server-Client and P2P architectures.

Both the architectures, showed promising results on the test set.

5.4 Varying Number of Clients

5.4.1 Performance

The performance (test set MSE loss) is compared for server and serverless architecture by varying the number of clients. The density of the network is kept at 0.5.

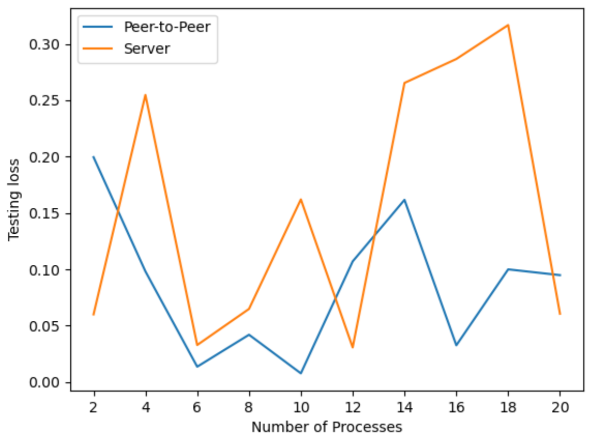

Figure 2: Testing loss vs Number of Processes

From figure 2, the test losses were almost the same for both server and serverless architectures. Even though the graph is 50% sparse, the serverless architecture gave good results; this may be due to the fact that the data is uniformly distributed across all clients. In majority of the cases, the serverless architecture's performance was one step ahead of the server architecture, this can be due to more communication in serverless architecture resulting in better model updates.

5.4.2 Communication Cost

The Communication cost is compared for server and serverless architecture by varying the number of clients. The density of the network is kept at 0.5.

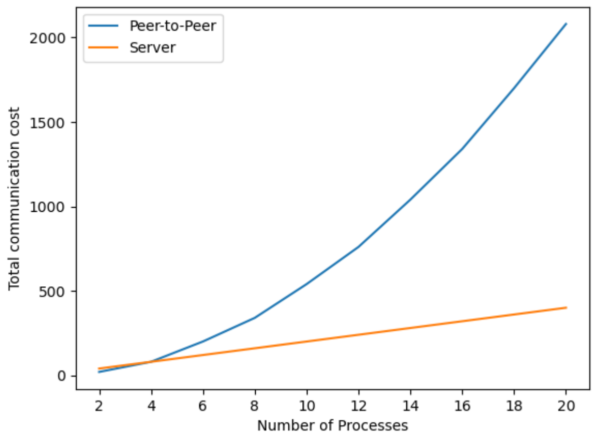

Figure 3: Communication Cost vs Number of Processes

From the figure 3, the communication cost is higher for the serverless case than the server case. This is expected as each process shares model information with all clients, whereas, in the server model, the clients share with only the central server. The increase in communication cost was almost quadratic in the case of the serverless model, whereas it is linear in the server model with an increase in the number of processes.

5.4.3 Time

The Time taken is compared for server and serverless architecture by varying the number of clients. The density of the network is kept at 0.5.

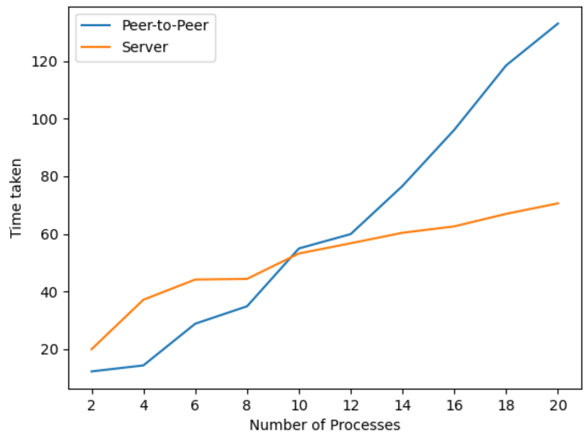

Figure 4: Time taken vs Number of Processes

From the figure 4, we can see that the Time taken is higher in serverless architecture than the server architecture. Initially, with a lower number of processes, the peer-to-peer model took lower times, but with an increase in the number of processes, it took longer times. The increase in time taken is greater for the serverless model than the server model. More Time taken for running the serverless architecture is due to synchronization delay (α-synchronizer) and more communications across clients.

5.5 Varying Density

For varying densities of the underlying P2P network, we used the following implementation. We construct a random spanning tree initially to ensure connectivity across all nodes, and then additional edges are added to adjust the density level. The density is varied between 0 and 1, 0 implies a tree, and 1 implies a fully connected network.

5.5.1 Performance

The performance (test set MSE loss) is analysed on serverless architecture by varying the density of the network. The number of processes is kept constant at 6.

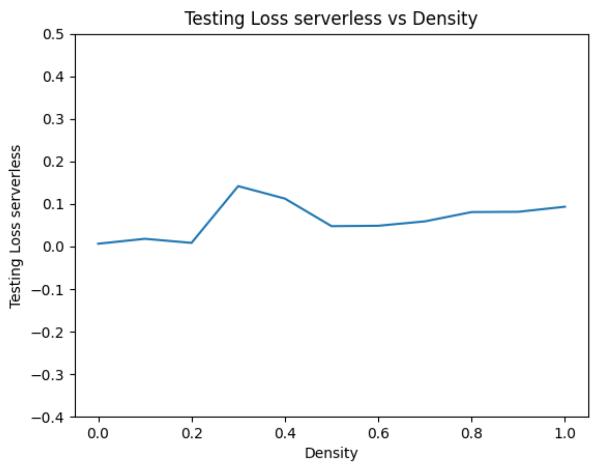

Figure 5: Testing Loss vs Density

From the figure 5, we can see that the test set loss is almost the same even with the increase in the complexity of the underlying P2P network. This can be due to the uniform distribution of data across all clients, so even if no information is passed from some of the clients directly, it doesn't affect the performance much.

5.5.2 Communication Cost

The Communication cost is compared for server and serverless architecture by varying the density of the network. The number of processes is kept constant at 6.

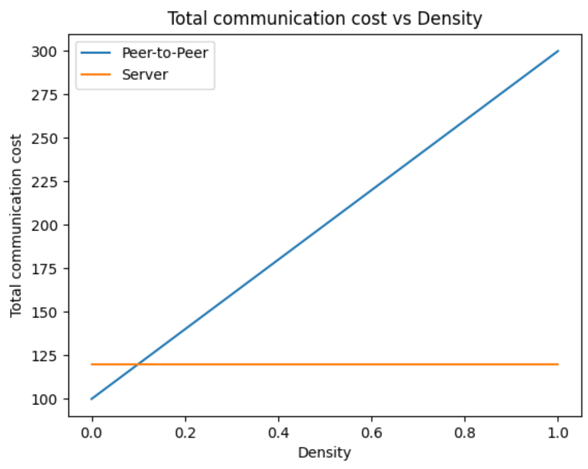

Figure 6: Communication cost vs density

From the figure 6, we can see that the communication costs in P2P architecture increase with an increase in the density of the network. This is due to an increase in the exchange of messages due to more connections in the network. However, For very sparse graphs (Density close to 0), we can see that the P2P has a slightly lower communication cost compared to the server-client architecture.

5.5.3 Time

The time taken is analysed using serverless architecture by varying the density of the network. The number of processes is kept constant at 6.

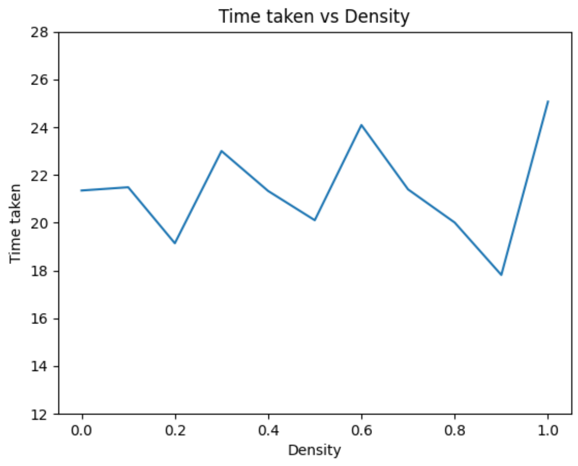

Figure 7: Time taken vs density

From the figure 7, we can see that the Time taken is almost the same even with the increase in the density of the underlying network. This is because each node broadcasts the messages to all its neighbours. However, the messages are passed through separate links between each pair of processes, so the transmission speed remains unaffected, leading to only a slight variation.

6 Drawbacks

However, shifting from server to decentralized peer-to-peer architecture has some disadvantages.

- From the results, it is quite evident that the communication costs were high in the case of peer-to-peer architecture, even with 50% sparsity.
- Compared to server-client architecture, in the case of P2P architecture, the synchronization is greater, leading to an increase in the algorithm's running Time.

7 Conclusion

In this project, we introduced algorithms for decentralized federated averaging for both synchronous and asynchronous systems. We simulated the proposed algorithms over a synthetic dataset. The results were analysed. By varying the number of clients and density of the graph, the changes in the communication complexity, time taken, performance on test set were compared and observations are made.

From the sparsity experiments, we can conclude that if the data is uniformly distributed across clients, even a very sparse network produced good results on test set with lower communication and synchronization overheads. We conclude that decentralized P2P systems are advantageous if the system is more prone to crashes. But this advantage comes with the overhead of increase in communication costs and synchronization overhead.

8 Possible Extensions

This work can be extended to use more sophisticated aggregation techniques present in the federated learning literature. So that the training could happen more smoothly even if the data is non i.i.d.

9 References

References

[1] Mohammad Mohammadi Amiri et al. *Federated Learning With Quantized Global Model Updates*. 2020. arXiv: 2006.10672 [cs.IT].

[2] Baruch Awerbuch. “Complexity of network synchronization”. In: *Journal of the ACM (JACM)* 32.4 (1985), pp. 804–823.

[3] H. Brendan McMahan et al. *Communication-Efficient Learning of Deep Networks from Decentralized Data*. 2023. arXiv: 1602.05629 [cs.LG].